Cooperation of Constraint Solvers: Using the New Process Control Facilities of ELAN

Peter Borovanský & Carlos Castro

LORIA-INRIA
615, rue du Jardin Botanique, BP 101,
54602 Villers-lès-Nancy Cedex, France
e-mail: {Peter.Borovansky,Carlos.Castro}@loria.fr

Abstract

In this paper, we introduce new process control facilities in the ELAN language as low-level primitives and high-level concurrent strategies. The low-level primitives are similar to those existing in UNIX for process control and communication. The high-level concurrent strategies proposed in this paper extend their existing sequential version giving them a new behaviour while preserving their operational semantics. This extension of the ELAN language allows to prototype in a very abstract and flexible way the integration of several computational systems within the same rewriting framework. Using the high-level concurrent strategies we can also avoid some non-terminating processes and/or apply the most efficient rewrite rule or strategy. We exemplify their use by prototyping different kinds of cooperation between constraint solvers, e.g. sequential and concurrent solvers.

Keywords: Rewriting Logic, Computational Systems, Constraint Satisfaction Optimisation Problems, Process Communication, Concurrent Strategies.

1 Introduction

The main motivation for this work is to extend the ELAN system with process control facilities, in order to allow an easy integration of several computational systems as needed for prototyping complex applications, e.g. cooperation of constraint solvers, and to allow parallelised executions to concurrent subderivations.

In declarative programming frameworks, like ELAN, a traditionally difficult question is how to integrate process control and Input/Output (I/O) operations in a declarative way. *A priori*, the communication with the external

[1] This work has been partially supported by the Esprit Basic Research Working Group 22457 - Construction of Computational Logics II.

world seems to be based on side-effects. There is a well-known approach used in functional programming languages based on the idea of *monads* introduced by Wadler [19]. However, this approach is not suitable for the first-order framework of rewriting, because monads are functional objects. Another approach integrating I/O operations, proposed by Viry, is based on the formalism of π-calculus [17]. An experimental implementation of this formalism has been integrated into the ELANsystem, but the final performance and usability of this system are not enough. In this paper, we propose an alternative approach based on low-level UNIX-like primitives and high-level concurrent strategies used for process control and communication. The low-level communication system offers I/O and process control facilities, which initialise (i.e. create), terminate (i.e. kill) and communicate (i.e. read and write via pipes) with processes in several ways. Furthermore, they are used in a non-declarative way. However, the communication with files is treated in the same manner as the communication with processes in a uniform and simple way.

The ELAN system has been developed as a framework for prototyping computational systems, that is, strategies controlling the application of a set of rewrite rules [11]. In previous works, we have described constraint solving as an inference process, where a constraint solver is viewed as a strategy specifying the order of application of the rules, in order to transform a set of constraints into a solved form [7,8]. To validate our approach, we have implemented the system COLETTE[2] which is currently executable in ELAN [9].

In the last twenty years, a lot of work has been done in the area of constraint solving and the need for powerful strategy languages controlling the integration of several independent solvers is now well understood. The cooperation between several constraint solvers enables to solve problems that cannot be tackled or efficiently solved with a single solver [1,13]. The paradigm of computational systems, as an integration of rule-based programming together with a flexible strategy language, seems to be of greatest interest for implementing the cooperation of several constraint solvers. That is why, in this paper, we illustrate the new ELAN process control facilities by prototyping the cooperation of constraint solvers.

The main contribution of this paper can be seen from two points of view. On one hand, we extend the existing strategy language of ELAN in order to prototype computational systems in a more flexible way. On the other hand, we use a rule-based approach for prototyping cooperation of solvers and provide a logical framework that could be useful for better understanding constraint solving.

This paper is organised as follows: section 2 briefly presents the ELAN language; section 3 presents the constraint solving problem we are interested in; section 4 introduces the new process control facilities of ELAN; and section 5 illustrates them by prototyping the cooperation of solvers. Finally, section 6

[2] `http://www.loria.fr/~castro/COLETTE/index.html`

concludes this paper and gives several perspectives for further research.

2 ELAN

In this section, we briefly present several features of the language used in this paper. A full description of ELAN can be found in [6]. Several short introductions are also presented in [5,4].

An ELAN program, as an instance of a computational system, is composed of conditional rewrite rules and strategies controlling derivations of these rewrite rules. These two parts can mutually refer to each other.

2.1 Rewrite Rules

ELAN rules are labeled conditional rewrite rules with local variable assignments of the following form:

$$[\ell] \quad l \Rightarrow r \quad [\ \textbf{if} \ \ v \mid \textbf{where} \ \ y{:=}(S)u\ ]^*$$

where

- ℓ is the label of the rewrite rule (eventually empty),
- l and r are the left and right-hand sides of the rule, respectively,
- v is a boolean condition,
- $y{:=}(S)u$ corresponds to the assignment of all results of the strategy S applied on a term u to the local variable y.

For applying such a rule on a term t, say at its top position, the left-hand side l is first matched against t. Then, the expressions introduced by **where** and **if** constructions are instantiated by the matching substitution. The instantiation of a local variable, such as y in the **where**-construction, extends the matching substitution. Therefore, **where** and **if** constructions of a rewrite rule are evaluated in textual order. When all conditions are satisfied and local assignments realised, the replacement of the left-hand side by the fully instantiated right-hand side is performed. In a more general case, non-linear patterns containing several variables composed of constructor symbols, e.g. $cons(y, cons(y, ys))$, can also be used in local assignments instead of single local variables.

Instantiations of local variables within **where** instructions may invoke ELAN strategies. A strategy S in a local assignment may be empty, which corresponds to the application of the default leftmost-innermost normalisation strategy built into the ELAN system. This normalisation strategy is parameterised by a set of unlabeled rewrite rules (i.e. when ℓ is empty), and it also respects OBJ-3 like local evaluation strategies associated to the function symbols of a signature. Labels of rewrite rules can be referred in the user's defined strategies.

Mainly for practical and efficiency reasons, ELAN offers a construction to

factorise conditional rewrite rules: several rewrite rules having the same label and left-hand side can be replaced by a rewrite rule where common parts to the right-hand side of each rewrite rule are specified once, and the different parts are specified using the operator **choose**. For example, the following rewrite rules:

$$[\ell]\ l \Rightarrow r \quad \textbf{where}\ y_1{:=}(S_1)u_1 \quad \textbf{if}\ v_2'(l, y_1) \quad \textbf{where}\ y_3{:=}(S_3)u_3$$
$$[\ell]\ l \Rightarrow r \quad \textbf{where}\ y_1{:=}(S_1)u_1 \quad \textbf{if}\ v_2''(l, y_1) \quad \textbf{where}\ y_3{:=}(S_3)u_3$$

can be replaced by:

$$\begin{array}{ll} [\ell]\ l \Rightarrow r & \textbf{where}\ y_1{:=}(S_1)u_1 \\ & \textbf{choose} \\ & \quad \textbf{try} \quad \textbf{if}\ v_2'(l, y_1) \\ & \quad \textbf{try} \quad \textbf{if}\ v_2''(l, y_1) \\ & \textbf{end} \\ & \textbf{where}\ y_3{:=}(S_3)u_3 \end{array}$$

We see that the common parts of the application of the first two rules, i.e. the matching of the left-hand side l and the assignment to the variable y_1 of the result of the application of S_1 on u_1, are executed only once in the factorised rule.

2.2 Strategies

An ELAN strategy is a way to describe non-deterministic computations in which the user is interested. An application of a rewrite rule in ELAN yields, in general, several results because of the **where** assignments, associative-commutative symbols, and application of several rewrite rules with the same label. When a rewrite rule or a strategy returns an empty set of results, we say that it *fails*. The language provides also a way to handle non-determinism. This is done using a strategy operator **dk**, standing for *don't know choose*. The strategy operator **dc**, standing for *don't care choose*, chooses randomly just one successful branch. Given a set of rewrite rules $\ell_1, \ldots, \ell_n$, the strategy:

- **dk** $(\ell_1, \ldots, \ell_n)$ returns *all* results of the applications of *all* rules $\ell_1, \ldots, \ell_n$,
- **dk one** $(\ell_1, \ldots, \ell_n)$ returns the *first* result of the application of each rule ℓ_i,
- **dc** $(\ell_1, \ldots, \ell_n)$ returns *all* results of the application of just *one* randomly chosen rule,
- **dc one** $(\ell_1, \ldots, \ell_n)$ returns the *first* result of the application of *one* randomly chosen rule,
- **first** $(\ell_1, \ldots, \ell_n)$ takes always the *first* successful rule ℓ_i in textual order and returns *all* its results,
- **first one** $(\ell_1, \ldots, \ell_n)$ returns just the *first* result of the *first* successful rule ℓ_i taken in textual order.

Each rewrite rule ℓ_i used by these strategy operators can be replaced either by a strategy expression or by a strategy identifier S_i, and the semantics remains

the same. A concatenation of strategies is expressed by the operator ";" and the repeated application of a strategy S, which iterates the strategy S until a normal form is obtained, is expressed by the strategy iterator **repeat***(S). The implementation of ELAN handles the non-determinism by an appropriate backtracking mechanism [15].

3 Constraint Satisfaction Optimisation Problems

In this section, we introduce the class of problems we are interested in, and that motivates the need for making solvers cooperate. The problem of finding values for a set of variables such that a set of constraints is satisfied is called a Constraint Satisfaction Problem (CSP) [12]. A CSP can be described by a tuple $\langle X, D, C\rangle$ representing the set of variables, the sets of possible values for the variables, and the set of constraints, respectively. Here, we concentrate on an extension of a CSP called Constraint Satisfaction Optimisation Problem (CSOP) that consists in finding an optimal (i.e. maximal or minimal) value for a given function, such that the set of constraints is satisfied [16]. In our opinion, the work of Bockmayr and Kasper [2] seems to be the best currently available reference explaining the approach generally used by the constraint solving community to deal with this problem. All details about the formalisation of CSOPs and the techniques for solving them, presented below, can be found in [10].

A CSOP can be described by a tuple $\langle P, f, lb, ub\rangle$ representing a CSP, an optimisation function, and the lower and upper bounds of this function, respectively. In this section, we illustrate two approaches for solving CSOPs. In section 5, we show how they can be combined using the new communication facilities and concurrent strategies of ELAN.

Without loss of generality, we consider the case of minimisation of a function f over integers. To deal with this problem, we consider two approaches, both of them requiring an initial step verifying that $Sol(C \wedge f \leq^? ub) \neq \emptyset$, i.e. there exists a solution of the set of constraints C respecting the additional constraint $f \leq^? ub$:

- Apply the following rule until it cannot be applied any more:

$$\langle P, f, lb, ub\rangle \;\rightarrow\; \langle P, f, lb, \alpha(f)\rangle \textbf{ if } \alpha \in Sol(C \wedge f <^? ub)$$

  Each iteration of this rule tries to decrease the upper bound ub by at least one unit until an unsatisfiable problem is obtained. That is why we call this technique *satisfiability to unsatisfiability*. The minimum value of the function f represents the upper bound of the last successful application of this rule.

- Apply the following rules until they cannot be applied any more:

$\langle P, f, lb, ub\rangle \rightarrow \langle P, f, lb, \alpha(f)\rangle$ **if** $\alpha \in Sol(C \wedge f <^{?} \frac{(lb+ub)}{2})$

$\langle P, f, lb, ub\rangle \rightarrow \langle P, f, \lceil\frac{(lb+ub)}{2}\rceil, ub\rangle$ **if** $lb \neq ub$ **and** $Sol(C \wedge f <^{?} \frac{(lb+ub)}{2}) = \emptyset$

  The first rule tries to find a new value for the upper bound ub and reduces, at least in half, the range of possible values of the function f each time a new solution is obtained[3]. The second rule similarly updates the lower bound lb in the opposite situation. We call this approach *binary splitting*.

In ELAN, we represent a CSOP using the following 4-tuple:

```
CSOP @    : (csp:Csp term:Term lb:int ub:int)  Csop;
```

The following strategy `MinSatToUnsat` implements the first approach:

```
stratop global
   MinSatToUnsat     : <Csop -> Csop>      builtin;
end
[.] MinSatToUnsat =>
      PostLTEUpperBound ;
      LocalConsistencyForEC ;
      GetNewUpperBound ;
      repeat* (first one (PostLTUpperBound ;
                          LocalConsistencyForEC ;
                          GetNewUpperBound
                          ,
                          SetLowerBoundToUpperBound)
              )
end
```

- The rule `PostLTEUpperBound` adds a constraint $x \leq^{?} ub$ to the set of constraints C. At the beginning of the solving process, we add a constraint $x =^{?} f$, where x is a new variable of this problem. We do it because once we post a unary constraint, the local consistency verification procedure can eliminate it. This is not true when we post a non-unary constraint.
- The strategy `LocalConsistencyForEC` tries to eliminate possible values for the variables while preserving the set of solutions of the current set of constraints[4].
- The rule `GetNewUpperBound` tries to find a solution for the current set of constraints. If it finds one, this rule updates the upper bound without modifying the set of the current constraints. Otherwise, this rule cannot be applied and the strategy `MinSatToUnsat` fails.
- The first sub-strategy tried by the strategy operator **first one** posts a con-

[3] Of course, we can think of different ratios, thus, the first approach can be seen as a particular case of the second one.

[4] This analysis is called *local consistency verification*. In our case, we verify *arc-consistency* for C. However, a detailed explanation is beyond the scope of this paper [7,8].

straint $x <^? ub$, verifies its local consistency (the set of constraints is eventually modified), and searches for a solution. By applying this sub-strategy, we obtain a new upper bound and a modified set of constraints (because of the local consistency verification). If this sub-strategy cannot be applied, because of the unsatisfiability of the problem adding a new constraint $x <^? ub$, the rule `SetLowerBoundToUpperBound` can be applied if the upper and lower bounds are still different. When these bounds are equal, this rule cannot be applied and the loop **repeat*** terminates.

The *satisfiability to unsatisfiability* approach does not use the value of the lower bound, and the rule `SetLowerBoundToUpperBound` is added just in order to keep the general structure of the second approach. In this way, we can reuse the same atomic rules for specifying both strategies.

The following strategy `MinSplitting` implements the *binary splitting* approach:

```
stratop global
   MinSplitting     : <Csop -> Csop>       builtin;
end
[.] MinSplitting =>
      PostLTEUpperBound ;
      LocalConsistencyForEC ;
      GetNewUpperBound ;
      EstimateLowerBound ;
      repeat* (first one (SetUpperBoundToMiddle ;
                          PostLTUpperBound ;
                          LocalConsistencyForEC ;
                          GetNewUpperBound ;
                          EstimateLowerBound
                          ,
                          SetLowerBoundToMiddle)
               )
end
```

- The operations before the loop **repeat*** are similar to those carried out by the previous strategy `MinSatToUnsat`. The only difference is that we add a step `EstimateLowerBound` computing the minimal value of the function f without considering the set of constraints, in order to update the lower bound whenever a new solution is found.
- The first sub-strategy tried by the strategy operator **first one** is similar to the first sub-strategy of the same operator in the strategy `MinSatToUnsat`, but there are some differences concerning the lower and upper bounds:
  - Before posting a constraint $x <^? ub$, we assign to the upper bound the value $\frac{(lb+ub)}{2}$ corresponding to the middle point of the interval of possible values for the function f if $lb \neq ub$.
  - Each time a new solution is found, we compute the smallest value that the function f can take, and assign this value to the lower bound.

· If the first sub-strategy of the strategy operator **first one** cannot be applied, we update the lower bound without verifying the existence of a solution giving a smaller value of the function f than the one corresponding to the middle point of the interval. We can avoid this verification because it has been done by the non-application of the first sub-strategy.

Concerning the behaviour of these strategies, we can note that:

- The strategy `MinSatToUnsat` takes a lot of time for reaching the minimal value of f, when it is located too far from the initial upper bound.
- Applying the strategy `MinSplitting`, the same situation happens when the minimal value of f is close to the initial upper bound.

Since it is not evident to know where the optimal solution is located, an *a priori* choice between these approaches is not possible in the general case.

4 The New Process Control Facilities in ELAN

We now describe several process control facilities proposed for the ELAN system as strategies at two levels. The high-level process control primitives are represented by strategies calling external processes, e.g. **dkcall**(P), or **dccall**(P), and by concurrent versions of the non-deterministic choice strategy constructors, e.g. **dk**(S_1 || ... || S_n), or **dc**(S_1 || ... || S_n). They concurrently execute all sub-strategies S_i, and ELAN as a dispatcher, collects all results of one or all sub-strategies, depending on the used strategy constructor. The low-level consists of several UNIX-like primitives for creation of processes and communication between them via pipes. These built-in primitives are made uniform with the I/O operations of ELAN. The high-level of process control facilities integrate these low-level operations into the existing strategy language of ELAN. In the next section, we illustrate some interesting applications of both levels.

4.1 Concurrent Strategies

The main motivation for introducing concurrent strategies in the ELAN system comes from practical considerations. For example, when we are interested in obtaining just one or all results of the application of some rewrite rule ℓ_i (or, some strategy S_i) using the strategy operators **dc**($\ell_1, \ldots, \ell_n$) or **dc one**($\ell_1, \ldots, \ell_n$), respectively, there exists the risk of never obtaining them. In the current sequential implementation of these operators, a rule ℓ_i may not terminate, and in this situation the whole computational process is indefinitely waiting. In practice, the user has to take care of this situation and, either be sure that all sub-strategies terminate, or try to apply them in an order ensuring that the whole process always terminates. Indeed, taking into account efficiency considerations, even if all sub-strategies terminate, the performance of the computational process depends on the order the strategies are tried.

When the results of all rules or sub-strategies are equivalent, i.e. there is no difference between them, the user would prefer to try the most efficient one firstly. Using the current strategy operators of ELAN, the only way the user can do this is by using the strategy operators **first** or **first one**. However, the order of strategies always depends on estimations of their efficiency. Based on these considerations, we have been interested in providing a more flexible way for specifying computational systems.

In order to express the concurrent execution of sub-strategies, we introduce the following basic strategies which correspond to the high-level of the process control constructions:

- concurrent choice of all results of all branches: **dk**,
- concurrent choice of one result of each branch: **dk one**,
- concurrent choice of all results of one branch: **dc**,
- concurrent choice of one result of one branch: **dc one**.

We also introduce basic strategy operators to control external processes:

- invocation of an external process for obtaining all its results: **dkcall**,
- invocation of an external process for obtaining one of its results: **dccall**.

The syntax and the operational semantics of these strategy operators are informally described as follows:

- **dk:** The strategy **dk** $(S_1 \mid\mid \ldots \mid\mid S_n)$ corresponds to a concurrent execution of all sub-strategies S_i for obtaining all results of all sub-strategies. It differs from its sequential version **dk**$(S_1, \ \ldots, \ S_n)$ in that the strategies $S_1, \ldots, S_n$ are applied simultaneously. If all strategies S_i fail, the strategy **dk** fails too.
- **dk one:** The strategy **dk one** $(S_1 \mid\mid \ldots \mid\mid S_n)$ corresponds to a concurrent execution of all sub-strategies S_i for obtaining just one result of each sub-strategy. The strategies $S_1, \ldots, S_n$ are applied simultaneously, and if all of them fail, the strategy **dk one** fails too.
- **dc:** The strategy **dc** $(S_1 \mid\mid \ldots \mid\mid S_n)$ corresponds to a concurrent execution of all sub-strategies S_i for obtaining all results of one successful sub-strategy. The implementation criteria currently chooses the strategy giving the first result. We have made this decision for efficiency reason, but we are aware of the risk of chosing a non-terminating rule or sub-strategy, or discarding a more efficient one. Another possible criteria is to chose the first strategy that gives all its results, which might be more expensive from the implementation point of view. Clearly, if all strategies S_i fail, the strategy **dc** fails.
- **dc one:** The strategy **dc one** $(S_1 \mid\mid \ldots \mid\mid S_n)$ corresponds to a concurrent execution of all sub-strategies S_i for obtaining the first result.
- **dkcall:** The strategy **dkcall** (P) corresponds to the execution of an external process, identified by P, for obtaining all its results. If the process returns an empty set of results, the strategy **dkcall** fails. The process P may

be a non-ELAN application (a solver) satisfying a simple communication protocol.

- **dccall:** The strategy **dccall** (P) corresponds to the execution of an external process, identified by P, for obtaining just one (i.e. the first) result.

All details about the strategy operators **dk**, **dk one**, **dc**, and **dc one** can be found in [3]. A more precise explanation with examples of the operators **dccall** and **dkcall** and their communication protocol is presented in [18].

Using the strategy primitives mentioned above, we can, for example:

- sequentially concatenate two processes: **dkcall**(P_1) ; **dkcall**(P_2),
- combine them with a sequential choice operator: **dk**(**dkcall**(P_1), **dkcall**(P_2)), or
- combine them with a concurrent choice operator: **dk**(**dkcall**(P_1) || **dkcall**(P_2)).

In these cases, the ELAN system maintains all running processes, i.e. it creates and kills processes, and communicates with them. However, for certain applications, e.g. those illustrated in section 5, we need to control processes in a specific and more fine way. In this case, the ELAN system offers several low-level primitives, which are more technical and less evident to use. However, using them, we can simulate the functionality of the strategy constructors described above, and discussed in the next section.

4.2 Low-level Process Control Primitives

We illustrate the possibility of controlling processes using several low-level primitives. These primitives are similar to operations existing in UNIX for process creation and communication. They have been integrated in a uniform way with the I/O system of ELAN, whose library module `stdio` contains four primitives for opening/closing files and for creating/killing processes. They have the following specification, where the sort Pid represents a process or file identification.

- The primitive **create**(`id`) creates a UNIX process identified by the string `id` as its name, and provides two blocking communication pipes:

$$\textbf{create}(@) \quad : \quad (string)\ Pid$$

- **create_noblock**(`id`) creates a process with two non-blocking pipes:

$$\textbf{create\_noblock}(@) \quad : \quad (string)\ Pid$$

- The primitive **open**(`id1`,`id2`) opens a file, identified by the string `id1`, with the modalities read/write/append as specified by the string `id2`:

$$\textbf{open}(@, @) \quad : \quad (string\ string)\ Pid$$

- The primitive **close**(`pid`) closes a file or kills a process identified by the term `pid` of the sort Pid:

$$\textbf{close}(@) \quad : \quad (Pid)\ Pid$$

When files are opened or processes created, the communication with them is realised using the communication primitives defined in the parameterised module `io[X]` of the ELAN library. The specification of these primitives is presented in the following. The parameter X stands for the sort of the data communicated, which is enriched by several error values as follows:

$$\textbf{Error}(@) \quad : \quad (int)\ X$$

- The primitive **write**(`pid`,`t`) writes a term `t` to a file or sends it to a process, identified by a term `pid`, via its input pipe. It returns the same term `t` or an error term **Error**(**errno**) if an error **errno** occurs:

$$\textbf{write}(@, @) \quad : \quad (Pid\ X)\ X$$

- The primitive **read**(`pid`) returns a term read from a file, or gotten from a process via its output pipe, identified by a term `pid`. If an error **errno** occurs, an error term **Error**(**errno**) is returned:

$$\textbf{read}(@) \quad : \quad (Pid)\ X$$

  In the case of non-blocking communication (i.e. a sub-process is created by **create_noblock**), whenever the father process reads a term from his sub-process, it is not blocked if there is nothing to read.

- The primitive **iserror**(`t`) returns `true` if `t` is an error term:

$$\textbf{iserror}(@) \quad : \quad (X)\ bool$$

  and the primitive **errno**(`t`) returns an error number encoded in the term `t`:

$$\textbf{errno}(@) \quad : \quad (X)\ int$$

  There are several error codes **errno** pre-defined in the module `io[X]`, e.g. `No_more` standing for the end of a file or a process communication, `No_term` saying that, currently, there is no term available in a non-blocking pipe, etc.

In the rest of this section, we develop a small example illustrating several different ways of the cooperation of two processes controlled by strategy-directed dispatchers written in ELAN. The motivations for this example are the following:

- to illustrate the low-level primitives, introduced above, on a small and slightly simplified example,
- to simulate some of the high-level concurrent strategies, introduced in the previous section, using these low-level primitives, and
- to discuss, in an abstract framework, several ways of cooperation between two processes, which are later (in section 5) instantiated by concrete constraint solvers for a CSOP.

We now introduce the following formalism of several simple strategies, in order to illustrate and model the behaviour of systems composed of several processes. Having two sub-processes $\mathtt{P_1}$ and $\mathtt{P_2}$, controlled by an ELAN dispatcher (their father), we introduce the following data structure of the sort `System`:

$$[\mathtt{pid_1}, \mathtt{pid_2} :: (\mathtt{in_0}, \mathtt{out_0}), \ldots, (\mathtt{in_2}, \mathtt{out_2})] : \mathtt{System}$$

where $\texttt{pid}_i$: Pid represents a process identification of $\texttt{P}_i$, and $\texttt{in}_i$, $\texttt{out}_i$ are lists of terms representing queues of terms of I/O pipes. Each sub-process $\texttt{P}_i$ communicates with its father by a pair of input and output pipes. The father can send terms to a sub-process $\texttt{P}_i$ putting them in its input queue $\texttt{in}_i$, or it can get results of a sub-process $\texttt{P}_i$ searching them in its output queue $\texttt{out}_i$.

The process $\texttt{P}_0$ represents the dispatcher (i.e. father) coordinating two sub-processes $\texttt{P}_1$ and $\texttt{P}_2$ (i.e. sons).

A set of basic and slightly simplified operations for specifying the communication of these processes is presented by the following rewrite rules:

```
[create i]    S => S[.pid_i<-pid]       where pid:=()create(P_i)
[close i]     S => S[.pid_i<-pid]       where pid:=()close(S.pid_i)
[get i]       S => S[.out_i<-S.out_i+x] where x:=()read(S.pid_i)
[send i]      S => S[.in_i<-xs]         where x.xs:=()S.in_i
                                        where y:=()write(S.pid_i,x)
[move i to j] S => S[.out_i<-xs][.in_j<-S.in_j+x]
                                        where x.xs:=()S.out_i
```

where S of the sort System represents the current state of the dispatcher $\texttt{P}_0$, and the symbol + stands for the concatenation of lists. The ELAN expression $\texttt{S.pid}_i$ means the sub-term of the structure S corresponding to the field $\texttt{pid}_i$, while the expression $\texttt{S}[.\texttt{pid}_i \leftarrow \texttt{pid}]$ replaces this field by its new value pid. All these rules work over the sort System, and labels of these rules define several simple (primal) strategies, which allow to explain different behaviours of concurrent systems.

As an example, we can illustrate the high-level concurrent strategies introduced in the previous section using these low-level strategies:

- The strategy `create i` creates a process $\texttt{P}_i$ and updates the corresponding field $\texttt{pid}_i$ in the structure S.
- A non-blocking version of the strategy `create` is called `create_noblock i`.
- The strategy `get i` reads a term from the output pipe of the process $\texttt{P}_i$ and puts it at the end of its output queue.
- The strategy `send i` takes the first term from the input queue of the process $\texttt{P}_i$ and sends it to the process $\texttt{P}_i$ via its input pipe.
- The strategy `move i to j` moves the first term from the output queue of the process $\texttt{P}_i$ to the end of the input queue of the process $\texttt{P}_j$.

We suppose that these rules fail whenever an operation is not permitted or produces an error. We can now illustrate, for example, the strategy $\texttt{S}_i$ = **dccall** ($\texttt{P}_i$) as a sequence of the following simple strategies searching for the first result of $\texttt{P}_i$:

```
create i; move 0 to i; send i; get i; move i to 0; close i
```

This sequence of strategies transforms the input state of the dispatcher represented by the structure $[?, ? :: (?, \texttt{t}), (?, ?), (?, ?)]$ into the following output

state $[?, ? :: (\mathtt{t}', ?), (?, ?), (?, ?)]$, where $\mathtt{t}'$ is a result of an application of the strategy $\mathtt{S_i}$ on the term $\mathtt{t}$, and the symbol ? means an unspecified value.

For the case of the strategy $\mathtt{S_i} = \mathbf{dkcall}\ (\mathtt{P_i})$, when we are looking for all results of the external process $\mathtt{P_i}$, the sequence is the following:

```
create i; move 0 to i; send i; repeat*(get i; move i to 0) ; close i
```

which transforms the input state $[?, ? :: (?, \mathtt{t}), (?, ?), (?, ?)]$ into the output state $[?, ? :: (\mathtt{t_1}.\cdots.\mathtt{t_n}, ?), (?, ?), (?, ?)]$, where $\{\mathtt{t_1}, \cdots, \mathtt{t_n}\}$ are all results of the application of the strategy $\mathtt{S_i}$ on the term $\mathtt{t}$.

The concatenation $\mathtt{S_1}$; $\mathtt{S_2}$ of strategies $\mathtt{S_1} = \mathbf{dkcall}\ (\mathtt{P_1})$ and $\mathtt{S_2} = \mathbf{dkcall}\ (\mathtt{P_2})$ can be simulated as follows:

```
create 1; move 0 to 1; send 1; repeat*(get 1; move 1 to 0); close 1;
repeat* (
 create 2; move 1 to 2; send 2; repeat*(get 2; move 2 to 0); close 2
)
```

which is inefficient because the dispatcher, i.e. the process $\mathtt{P_0}$, waits for all results of the process $\mathtt{P_1}$, while the second process $\mathtt{P_2}$ is not active. This can be improved by the following sequence:

```
create 1; create 2; move 0 to 1; send 1;
  repeat* (get 1; move 1 to 2; send 2);
  repeat* (get 2; move 2 to 0);
close 1; close 2
```

where the dispatcher $\mathtt{P_0}$ sends all results of the process $\mathtt{P_1}$ to the second process $\mathtt{P_2}$ as soon as they are obtained.

Now, let us look at the case of concurrently launched processes controlled by the dispatcher.

We formulate this problem as follows: a sequence of terms $\mathtt{t_1} \ldots \mathtt{t_n}$, $\mathtt{n} > 1$, should be reduced by a strategy $\mathtt{S} = \mathbf{dccall}\ (\mathtt{P})$ in order to obtain a sequence of their results $\mathtt{r_1} \ldots \mathtt{r_n}$, i.e. $\mathtt{r_i}$ is a result of the application of the strategy $\mathtt{S}$ on a term $\mathtt{t_i}$. In order to obtain these results more rapidly, we can concurrently launch several copies of the same process, which are controlled by the dispatcher. For simplicity, we suppose here that $\mathtt{S_1} = \mathtt{S_2} = \mathtt{S}$.

```
create 1; create 2; move 0 to 1; send 1; move 0 to 2; send 2;
  repeat* (
    get 1; move 1 to 0; move 0 to 1; send 1;
    get 2; move 2 to 0; move 0 to 2; send 2
  );
close 1; close 2
```

In this approach, the input terms are distributed one-by-one to both subprocesses. It is clear that waiting for a result of one process can block the other process. Therefore, we propose a non-blocking approach, where the operation $\mathtt{get_i}$ does not block the execution of the dispatcher. During the execution of sub-processes, the dispatcher is idle, or it makes his own work.

```
create_noblock 1; create_noblock 2;
move 0 to 1; send 1; move 0 to 2; send 2;
repeat* (
    first (get 1; move 1 to 0; move 0 to 1; send 1,
           get 2; move 2 to 0; move 0 to 2; send 2,
           idle
          )
)
close 1; close 2
```

In the next section, the formalism sketched in this section is adapted to prototype several kinds of cooperation between constraint solvers of a CSOP.

5 Cooperation of Constraint Solvers

Informally, cooperation of constraint solvers involves communication problems between solvers devoted to a single domain [14]. In order to improve the performances of two basic solvers presented in section 3, we present several approaches for making them cooperate.

The only difference between the strategies `MinSatToUnsat` and `MinSplitting` presented in section 3 is in the loop **repeat***. However, both strategies carry out the same initialisation step. We can factorise this initialisation step and obtain the following version of these strategies:

- The strategy `SetUpperAndLowerBounds` carries out the common initialisation step of both strategies:

  ```
  [.] SetUpperAndLowerBounds =>
        PostLTEUpperBound ;
        LocalConsistencyForEC ;
        GetNewUpperBound ;
        EstimateLowerBound
  end
  ```

- The strategy `MinSatToUnsat` is then expressed in the following way:

  ```
  [.] MinSatToUnsat =>
        SetUpperAndLowerBounds ; repeat* (strat1)
  end
  ```

  where the sub-strategy `strat1` describes operations updating of the lower and upper bounds carried out in each iteration step of the loop **repeat***.

  ```
  [.] strat1 =>
        first one (PostLTUpperBound ;
                   LocalConsistencyForEC ;
                   GetNewUpperBound
                   ,
                   SetLowerBoundToUpperBound)
  end
  ```

- The strategy `MinSplitting` is then expressed in the following way:

```
[.] MinSplitting =>
      SetUpperAndLowerBounds ; repeat* (strat2)
end
```

  where the strategy `strat2` describes the basic step that is carried out in each iteration step as follows:

```
[.] strat2 =>
      first one (SetUpperBoundToMiddle ;
                 PostLTUpperBound ;
                 LocalConsistencyForEC ;
                 GetNewUpperBound ;
                 EstimateLowerBound
                 ,
                 SetLowerBoundToMiddle)
end
```

As we have explained in section 3, the distance between the initial upper bound and the localisation of the optimal solution has a strong influence in the performance of these strategies. Taking these considerations into account, it seems to be a good idea to make the sub-strategies `strat1` and `strat2` cooperate, in order to profit from the advantages of each one, and to avoid their drawbacks.

In the following, we present several cooperation schemes using these strategies as elementary solvers. A first scheme of the cooperation between the solvers `strat1` and `strat2` is expressed by the strategy `CooperationI`:

```
[.] CooperationI =>
     SetUpperAndLowerBounds ;
     repeat* ( strat1 ; first one (strat2 , id))
end
```

Using the strategy `CooperationI`, the whole work is done by the same computational process. In the next example, we design these solvers as external processes. We define two ELAN programs, called `solver1CSOP` and `solver2CSOP`, applying the strategies `strat1` and `strat2`, respectively. The strategy `CooperationII` implements this idea:

```
[.] CooperationII =>
      SetUpperAndLowerBounds ;
      repeat* (dccall (solver1CSOP) ;
               first one (dccall (solver2CSOP) , id))
end
```

The behaviour of the strategies `CooperationI` and `CooperationII` is similar because both solvers are executed sequentially. In [14], this approach is called *sequential solvers*. Its obvious disadvantage is leaving a solver inactive, while the other one is working. Moreover, due to the exponential complexity of the problem under consideration, the whole process could be blocked if one solver

cannot find a solution. To avoid this situation, we can think of running them concurrently, updating the current solution as soon as a new one is available, and stopping the other solver. In [14], this approach is called *concurrent solvers*, and the following strategy, `CooperationIII`, implements this kind of cooperation:

```
[.] CooperationIII =>
      SetUpperAndLowerBounds ;
      repeat* (dc one (strat1 || strat2))
end
```

Of course, using the strategy `CooperationIII`, the whole work is done by the same computational process. We could think again of using the solvers as external processes, as done by the following strategy, `CooperationIV`:

```
[.] CooperationIV =>
      SetUpperAndLowerBounds ;
      repeat* (dc one (dccall (solver1CSOP) || dccall (solver2CSOP)))
end
```

With the strategies `CooperationIII` and `CooperationIV`, a solver never waits for a solution coming from the other one. When all solutions are read from the same elementary solver until the final solution is obtained, the performance of these new solvers, `CooperationIII` and `CooperationIV`, is the same as if one of the elementary solvers runs independently.

The strategies `CooperationIII` and `CooperationIV` avoid blocking the solvers because they are re-launched whenever a new solution is found. In some situations, a process running slowly, but eventually returning a better solution than the one generated by the quicker solver, can be killed. To avoid this situation, we could think of running the solvers in parallel, and updating the current CSOP as soon as a new solution is available, without stopping the other solver, no matter if two or more consecutive solutions come from the same solver. In this case, we hope that the slower solver can contribute with more information. To implement this cooperation, we introduce the following data structure:

```
CSOPbis @    : (csop:Csop pids:list[Pid])  Csopbis;
```

The first component stores the current solution of the CSOP, and the second one stores a list containing the identification of the running processes.

The rewrite rule `CreateSolver1NoBlock` creates the external process called `solver1CSOP` implementing the strategy `strat1`. The process identification `pid`, resulting from the application of the operator `create_noblock`, is added to the list of running solvers `lPid`. The operator `create_noblock` creates a non blocking process. In the same way, we define the rule `CreateSolver2NoBlock` to create the external process called `solver2CSOP`, whose `pid` is added to the list `lPid`.

```
[CreateSolver1NoBlock]
   CSOPbis[P1,lPid] => CSOPbis[P1,pid.lPid]
      where   pid := ()create_noblock("solver1CSOP")
      if      not iserr(pid)
```

Once the solvers are launched, they stay idle. To activate the solvers, we define the rule `SendToSolver` that takes the solver identification and sends the current CSOP to it:

```
[SendToSolver]
   CSOPbis[P1,pid.lPid] => CSOPbis[P1,pid.lPid]
      where   P2      := ()write(pid,P1)
      where   string1 := ()write(pid,"\nend\n")
      if      not iserror(P2) and not iserror(string1)
```

In the previous examples, a new solution is always better than the current one, because the current solution was given as the input to the solver. Now, we have to verify the quality of the new solution in order to decide the updating of the current solution. That is why, we define the following rule `GetOutputOfSolver`:

```
[GetOutputOfSolver]
   CSOPbis[CSOP[P,x,lb,ub],pid.lPid] => CSOPbis[R,pid.lPid]
      if    lb != ub           // The current solution is not optimal
      where  Q :=()read(pid)   // Read from the pipe
      if     Q != No_term      // A new solution is available
      choose
        try
           if GetLowerBound(Q) > lb or GetUpperBound(Q) < ub
           where R := ()Q      // Update of the current solution
        try
           where R := ()CSOP[P,x,lb,ub] // The current solution
      end                               // is better
```

Once a new solution has been obtained, we put the process identification `pid` at the end of the list of processes. We also do that, when the first solver in the list has not yet returned any result. This criteria controls the list of solvers and it is expressed by the following rule `ChangePriority`:

```
[ChangePriority]
   CSOPbis[CSOP[P,x,lb,ub],pid.lPid] =>
   CSOPbis[CSOP[P,x,lb,ub],append(lPid,pid.nil)]
      if      lb != ub
```

Finally, we kill a process using the following rule `CloseSolver`:

```
[CloseSolver]
   CSOPbis[P1,pid.lPid]  =>  CSOPbis[P1,lPid]
      where pid1 := ()close(pid)
```

The strategy CooperationV implements the last approach as follows:

```
[.] CooperationV =>
      SetUpperAndLowerBounds ;
      CreateSolver1NoBlock ;
      CreateSolver2NoBlock ;
      repeat* (SendToSolver) ;
      repeat* (first one (GetOutputOfSolver ;
                          SendToSolver ;
                          ChangePriority
                          ,
                          ChangePriority)
              ) ;
      repeat* (CloseSolver)
end
```

Of course, an *a priori* choice among all these possibilities of cooperation is impossible in the general case. Their performance, and the performance of new ones, should be analysed in specific cases. This is beyond the scope of this paper. Our interest is just to exemplify how we can prototype in a very flexible and abstract way different kinds of cooperation of constraint solvers using the new communication facilities of ELAN. Considering that constraint solving is a strongly heuristics activity, the possibility of using several solvers in a way independent of the order of their specification is evidently interesting. All examples presented in this paper have been integrated into the system COLETTE.

6 Conclusions

We have introduced new process control facilities in the ELAN language at two levels: the low-level consists of several UNIX-like primitives and the high-level consists of concurrent strategies whose semantics is similar to their sequential version. This extension of the language allows a more flexible prototyping of computational systems in the same rewriting framework. Using the high-level strategies, we have been able to easily prototype the sequential and concurrent cooperation of solvers in a very abstract way. Using the low-level facilities, we have prototyped a more complex kind of cooperation. This work can be considered a first attempt to implement the cooperation of constraint solvers using the rewriting framework. It seems to be interesting to take into account detailed information about each solver when controlling them (execution times, statistics about efficiency, etc.) and much more work should be done in that direction. The low-level primitives represent a possibility harder to use, less intuitive, but in many cases more general. The high-level concurrent strategies offer an easier, clear, and declarative way for programming, but not as general as the previous one. The gap between these two extremes should be attacked in future research in order to combine the advantages of both approaches.

Acknowledgements

We are grateful to Hélène Kirchner, Claude Kirchner, and Christophe Ringeissen for providing useful comments on an earlier version of this paper, and to the anonymous referees for their remarks that allowed us to improve this work.

References

[1] H. Beringer and B. DeBacker. Combinatorial Problem Solving in Constraint Logic Programming with Cooperative Solvers. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence. North Holland, 1995.

[2] A. Bockmayr and T. Kasper. A unifying framework for integer and finite domain constraint programming. Research Report MPI-I-97-2-008, Max Planck Institut für Informatik, Saarbrücken, Germany, August 1997.

[3] P. Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, October 1998.

[4] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, Japan, April 1998. World Scientific.

[5] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In *Proceedings of The Second International Workshop on Rewriting Logic and its Applications, WRLA'98*, Pont-à-Mousson, France, September 1998. Electronic Notes in Theoretical Computer Science.

[6] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. *ELAN Version 3.00 User Manual.* CRIN & INRIA Lorraine, Nancy, France, first edition, January 1998.

[7] C. Castro. Solving Binary CSP Using Computational Systems. In J. Meseguer, editor, *Proceedings of The First International Workshop on Rewriting Logic and its Applications, RWLW'96*, volume 4, pages 245–264, Asilomar, Pacific Grove, CA, USA, September 1996. Electronic Notes in Theoretical Computer Science. Also available as technical report 96-R-190 of the Centre de Recherche en Informatique de Nancy, CRIN.

[8] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3):263–293, June 1998.

[9] C. Castro. COLETTE, Prototyping CSP Solvers Using a Rule-Based Language. In J. Calmet and J. Plaza, editors, *Proceedings of The Fourth International*

Conference on Artificial Intelligence and Symbolic Computation, Theory, Implementations and Applications, AISC'98, volume 1476 of *Lecture Notes in Artificial Intelligence*, pages 107–119, Plattsburgh, NY, USA, September 1998.

[10] C. Castro. *Une approche déductive de la résolution de problèmes de satisfaction de contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, 1998. To appear.

[11] C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. V. Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers*, pages 131–158. The MIT press, 1995.

[12] A. K. Mackworth. Constraint Satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992. Second Edition.

[13] P. Marti and M. Rueher. A Distributed Cooperating Constraints Solving System. *International Journal of Artificial Intelligence Tools*, 4(1-2):93–113, 1995.

[14] E. Monfroy. *Collaboration de solveurs pour la programmation logique à contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, November 1996. Also available in english.

[15] P.-E. Moreau and H. Kirchner. A Compiler for Rewrite Programs in Associative-Commutative Theories. In *Proceedings of Programming Language Implementation and Logic Programming, PLILP'98*, Lecture Notes in Computer Science. Springer-Verlag, September 1998. To Appear.

[16] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[17] P. Viry. Input/Output for ELAN. In J. Meseguer, editor, *Proceedings of The First International Workshop on Rewriting Logic and its Applications, RWLW'96*, volume 4, pages 51 – 64. Electronic Notes in Theoretical Computer Science, September 1996.

[18] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, October 1994.

[19] P. Wadler. Monads for functional programming. In E. Meijer and J. Jeuring, editors, *Proceedings of Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, 1995.